"Achieving Century Uptimes"
An Informational Series on Enterprise Computing

As Seen in *The Connection*, A Connect Publication
December 2006 – Present

About the Authors:

Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, have a combined experience of over 90 years in the implementation of fault-tolerant, highly available computing systems. This experience ranges from the early days of custom redundant systems to today's fault-tolerant offerings from HP (NonStop) and Stratus.

Gravic, Inc.
Shadowbase Products Group
17 General Warren Blvd.
Malvern, PA 19355
610-647-6250
www.ShadowbaseSoftware.com

Achieving Century Uptimes

Part 17: HP Unveils Its Synchronous Replication API for TMF

July/August 2009

Dr. Bill Highleyman
Paul J. Holenstein
John R. Hoffmann

In our earlier "Achieving Century Uptimes" article in the November/December issue of *The Connection* ("Part 13: Synchronous Replication: Pros, Cons, and Myths"), we compared asynchronous and synchronous replication. Synchronous replication avoids the problems of data loss and data collisions associated with asynchronous replication at the expense of adding application latency as the transaction commits across the network. Synchronous replication works by including all of the distributed copies of the data to be modified within the scope of a single transaction.

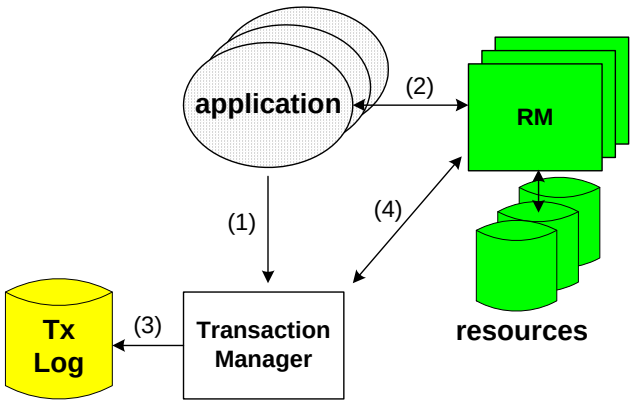
The problem is that there have been no products offering synchronous-replication engines on NonStop systems. NonStop's Transaction Management Facility (TMF) did not allow third-party software to participate in the commit phase of the transaction, which is required to allow efficient, off-the-shelf, synchronous replication. That is now changing. HP has recently announced its Synchronous Replication Gateway (SRG - code-named "Open TMF", or OTMF, during development) that allows a foreign resource manager, such as a replication engine, to join a TMF transaction. SRG follows the X/Open model for distributed transactions. However, its API is different.

The X/Open Model

The two-phase commit protocol (2PC) is defined as a standard by the X/Open Group for Distributed Transaction Processing (DTP).[1] It is often referred to as the X/Open DTP specification.

As shown in Figure 1, the X/Open DTP model comprises five components:

- application programs.
- a Transaction Manager (TM).
- resources such as disks, queue managers, or applications.
- Resource Managers (RMs) that hide the attributes of resources.
- a transaction log (Tx Log).



The X/Open Distributed Transaction Processing Model
Figure 1

Applications use resources such as

[1] Distributed Transaction Processing: The XA Specification, *The Open Group*; 1991.

databases or queues. Each resource is managed by a Resource Manager (RM). The RMs hide the details of their resources from the applications and from the Transaction Manager (TM) by providing a common interface used by the other components.

When an application begins a transaction (1), the TM assigns the transaction an ID and monitors its progress, taking responsibility for its success or failure. All changes to a resource such as to a database (2) are typically logged in a transaction log by the TM (3). The transaction log provides the change information necessary should a transaction abort and need to be backed out or should a system fail and its database require reconstruction.

The TM has a particularly important responsibility at commit time. When the application directs the TM to commit a transaction (1), the TM first queries each RM to ensure that it is prepared to commit the transaction (4). This is Phase 1 of the two-phase commit protocol, the *prepare* phase. An RM will respond positively to the prepare query (that is, it *votes "yes"* for the commit) only if it has safe-stored or has tentatively applied the change data for the transaction to the target database, thereby assuring that the RM can ultimately make all of the transaction's changes to the database.

If all RMs reply that they can commit the transaction, the TM issues a commit directive to all RMs (4). This is Phase 2 of the 2PC, the *commit* phase. When an RM receives a commit directive, it commits the changes to the database.

Alternatively, if any RM votes "no" because it cannot make the transaction's changes, the TM issues an abort directive to all RMs. Each RM either makes no changes, or it rolls back the transaction's changes if it has already tentatively applied them to the database. The transaction has no effect on the database.

Volatile-Resource Managers

Normally, a resource manager not only participates in the 2PC protocol, but it also engages in the TM's recovery process. The recovery process is used to recover the database following a failure that may have left the database in a corrupted or inconsistent state. However, there is a class of resource managers called *volatile-resource managers (VRMs)* that participate in the 2PC protocol but not in the recovery process. A VRM manages a volatile resource that is assumed to be nondurable and therefore does not benefit from recovery. In SRG, foreign resource managers are treated as VRMs. If they, in fact, are durable, they are responsible for their own recovery.

A synchronous-replication engine that participates in TMF transactions is treated as a VRM. TMF enforces this so that a third-party application cannot prevent recovery after a failure and hence keep the database offline.

The SRG API

SRG is a library that allows a third-party gateway process to interface with TMF through a VRM that has been implemented as a state machine within TMF. The library uses a Resource Manager pseudo-file (RM file) to identify the VRM and to exchange TMF signals (messages)

with the VRM via library API calls and several standard Guardian procedures. A gateway process may have multiple RM files open; this can be useful if the gateway must manage more than one simultaneous transaction.

TMF signals include, among others:

- TMF_SIGNAL_EXPORT_DONE: A no-waited request to the VRM from the gateway to join a transaction has completed.

- TMF_SIGNAL_REQUEST_PREPARE: The VRM is requesting that the gateway vote on the outcome of the transaction.

- TMF_SIGNAL_READY: The gateway is indicating to the VRM that the gateway's transaction is prepared to commit.

- TMF_SIGNAL_REQUEST_COMMIT: The VRM is indicating to the gateway that the transaction has been committed.

- TMF_SIGNAL_REQUEST_ROLLBACK: Either the VRM or the gateway is indicating that the transaction should be aborted. The signal also carries the reasons for the abort.

- TMF_SIGNAL_FORGET: The gateway is informing the VRM that the gateway has completed processing the transaction.

There are other signals to indicate that TMF is enabled, disabled, or down (crashed).

API

The SRG API is simple. It contains only six calls.

- OTMF_VOL_RM_OPEN: Opens a VRM file. A VRM file must be open before the gateway process can communicate with the VRM. As soon as the file is opened, the gateway is informed as to whether TMF is enabled, disabled, or down. The file can be closed via a standard file close call.

- OTMF_EXPORT: Allows the gateway to participate in a transaction.

- OTMF_EXPORT_ASYNC: Allows the gateway to make a no-waited request to the VRM to participate in a transaction. The request's completion is indicated by the receipt of a TMF_SIGNAL_EXPORT_DONE signal.

- OTMF_WRITE_SIGNAL: Used by the gateway to send a signal to the VRM via the RM file.

- OTMF_WAIT_SIGNAL: Waits for a signal from a VRM following a READX on the RM file.

- OTMF_INTERPRET_SIGNAL: Interprets a signal returned in a call to AWAITIOX instead of a call to OTMF_WAIT_SIGNAL. Instead of calling OTMF_WAIT_SIGNAL and waiting for the signal, the gateway can call AWAITIOX instead, allowing other I/O completions to be processed as they complete. OTMF_INTERPRET_SIGNAL is then called to parse the data sent by the VRM.

SRG and Coordinated-Commit Replication

Let us use the coordinated-commit protocol[2] as an example to illustrate the application of the SRG API to synchronous replication.[3] An asynchronous-replication engine suffers from the possibility of lost data following a source-node failure and from data collisions when running in an active/active environment. A synchronous-replication engine suffers from increased transaction-response time due to application latency as it waits for each operation to complete across the network. A coordinated-commit replication engine is an interesting combination of asynchronous and synchronous replication technology that eliminates data loss and data collisions and minimizes application latency.

The Coordinated-Commit Protocol

With coordinated commits, when the application starts a transaction, the replication engine participates in the transaction via the SRG API through a VRM. Asynchronous replication is used to replicate updates to the target database, locking the data objects as it does so. Thus, there is no additional latency imposed upon the application during this process. However, transaction commit is synchronous. As a result, no data is lost should the source node fail. Likewise, since all data objects are locked at both the source and target databases until commit time, there can be no data collisions. Application latency only occurs as the source system waits for the replication engine to vote rather than after every update that has been issued by the source system, as in the case of the dual writes approach.

The coordinated-commit protocol is shown in Figure 2 as it would be implemented on a NonStop server. The coordinated-commit replication engine is a VRM gateway. Both TMF and the replication engine are informed when the application begins a transaction (1). At this point, the replication engine requests that it join the transaction (2). This lets it vote on the outcome.

As the application issues updates (3a), they are written to the DP2 disk processes (3b) that are the RMs for the NonStop disks. Updates are also written to the TMF audit log. The

[2] B. D. Holenstein, P. J. Holenstein, G. E. Strickler, Collision avoidance in data replication systems, *U. S. Patent 7,103,586*; September 5, 2006.

B. D. Holenstein, P. J. Holenstein, W. H. Highleyman, Asynchronous coordinated commit replication and dual write with replication transmission and locking of target database on updates only, *U. S. Patent 7,177,866*; February 13, 2007.

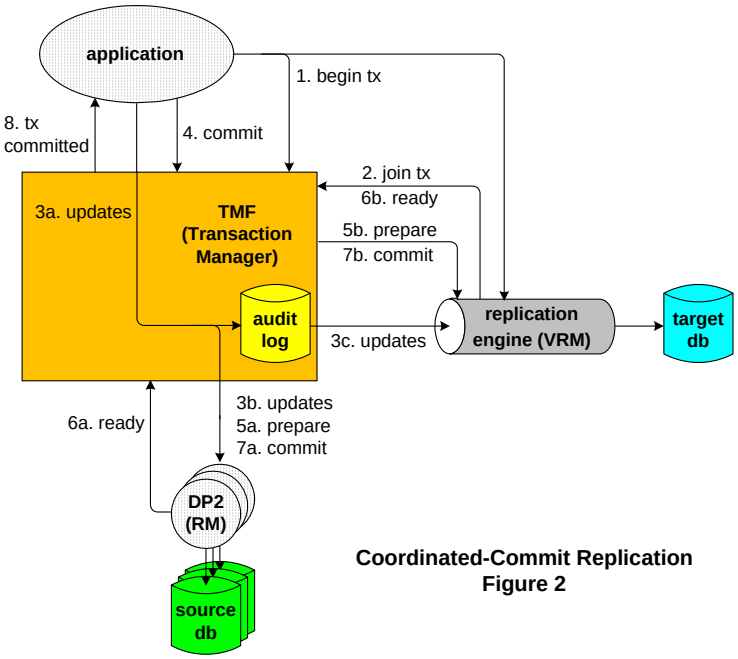
[3] Chapter 4, Synchronous Replication, *Breaking the Availability Barrier***,** AuthorHouse; 2004.

Synchronous Replication, *Availability Digest*; December, 2006.

Synchronous Replication: Pros, Cons, and Myths, *The Connection*; November/December, 2008.

replication engine reads the updates from the audit log (3c) and replicates them to the target database, where they are tentatively applied.

When the application issues a Commit directive (4), TMF sends Prepare signals in parallel to all of its Resource Managers (5a), including the VRMs (the replication engine in this case) (5b). This is the Prepare phase of the two-phase commit protocol. The Resource Managers check that they have safe-stored or tentatively applied all updates within the scope of the transaction and if so reply with a "yes" vote - a Ready signal (6a, 6b) - to TMF. If all RMs have voted "yes," TMF sends a Commit signal (7a, 7b) to all RMs – the Commit phase - and notifies the application that its transaction has been committed (8).

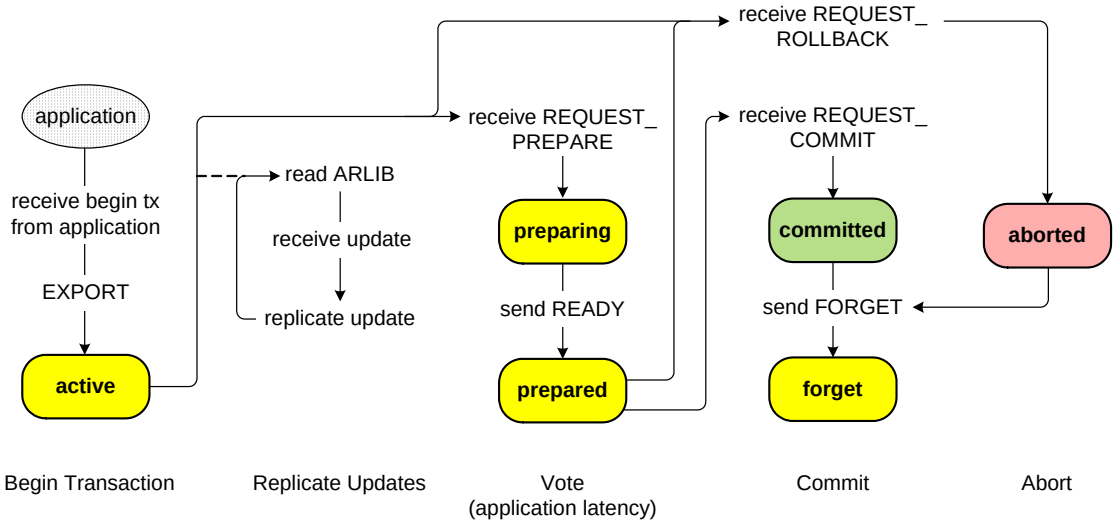


Coordinated-Commit Replication
Figure 2

If any RM cannot commit the transaction, it votes "no;" and TMF will send a Rollback signal to all RMs, informing them to abort the transaction.

Mapping Coordinated-Commit Replication to the SRG API

The use of the SRG API and signals to implement a coordinated-commit replication engine in a NonStop environment is shown in Figure 3. In this figure, "send" means sending a signal via



The Use of the OTMF API by a Coordinated-Commit Replication Engine
Figure 3

OTMF_WRITE_SIGNAL, and "receive" means receiving a signal via a READX/ OTMF_WAIT_SIGNAL or a READX/AWAITIOX/OTMF_INTERPRET_SIGNAL sequence.

Begin Transaction

When an application issues a begin transaction, it notifies the coordinated-commit replication engine via an interface provided by the replication engine. The replication engine joins the transaction via the OTMF_EXPORT or OTMF_EXPORT_ASYNC API call. At this point, the VRM enters the *active* state.

Replicate Updates

While in the *active* state, the replication engine asynchronously replicates updates by extracting changes from the TMF audit trail via ARLIB, the Audit Reader Library provided by TMF. Each update is buffered and sent over the replication channel to the target database, where it is tentatively applied.

Vote (the Prepare Phase)

When the replication engine receives a TMF_SIGNAL_REQUEST_PREPARE signal from TMF, it enters the *preparing* state. It waits until it can confirm that all of the updates within the scope of the transaction have been tentatively applied to the target database, and it then returns a TMF_SIGNAL_READY signal to TMF. This delay is the application latency added by coordinated-commit replication. At this point, the replication engine enters the *prepared* state. If the replication engine is unable to commit the transaction at the target database, it returns a TMF_SIGNAL_REQUEST_ROLLBACK signal instead, which causes the transaction to be aborted (not shown in Figure 3).

Commit/Abort (the Commit Phase)

When the replication engine receives a TMF_SIGNAL_REQUEST_COMMIT or TMF_SIGNAL_REQUEST_ROLLBACK signal in the *prepared* state, it can initiate the appropriate target-side processing and immediately send a TMF_SIGNAL_FORGET signal to complete the source-side transaction.

Summary

The SRG API allows TMF to safely support gateways to foreign systems through volatile-resource managers. This capability allows replication engines to be integrated with TMF so that updates to remote databases can be synchronously replicated. Check with your replication partner for their plans in this area.